Introduction

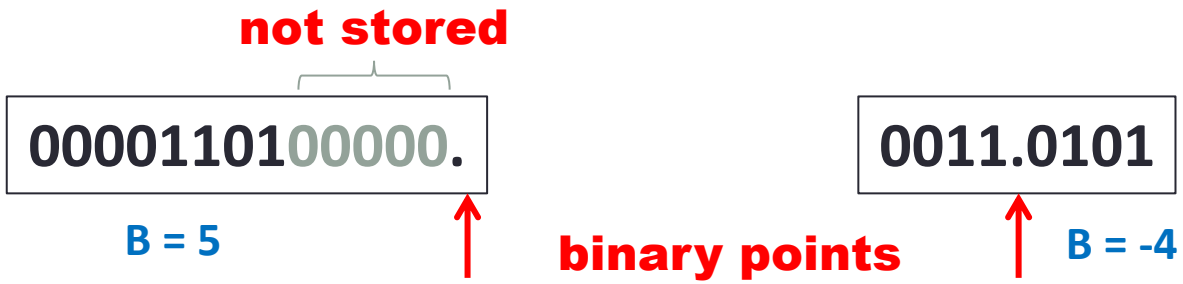
- How to represent fractional numbers
  - Fixed point
  - IEEE-754 floating point (32 bit)
- Symbolic walk-through in CT1 & CT2
- Loops (CT3 preview)
  - Conditional branches after CMP
  - Analysing loops
- CT3 Challenge (optional)
- ARM instruction quick reference

Arithmetic on real numbers 1: Fixed Point

- So far, we have concentrated on **integer** representations – signed or unsigned.
- There is an implicit binary point to the right:

00001101.

- In general, the binary point can be in the middle of the word (or off the end!). This is a FIXED POINT representation of real numbers



- Fixed point arithmetic requires no extra hardware – the binary point is in the mind of the programmer, like signed/unsigned the CPU does not care!
- What is the **resolution** of the two fixed point schemes above?
- Fixed point works equally well with signed and unsigned numbers
  - Fixed point value = integer value * 2^B
  - B is 0 - number of fractional bits to RIGHT of binary point
  - B is number of added zeros to LEFT of binary point

B is not known by CPU
In mind of programmer

Arithmetic on Real Numbers 2: Floating Point

- Although fixed point representation can cope with numbers with fractions, the range of values that can represented is still limited.
- Alternative: use the equivalent of "scientific notation", but in binary:

$$\text{number} = s \times m \times 2^e$$

sign mantissa exponent

For example:

10.5 decimal in binary: $1010.1_{(2)}$

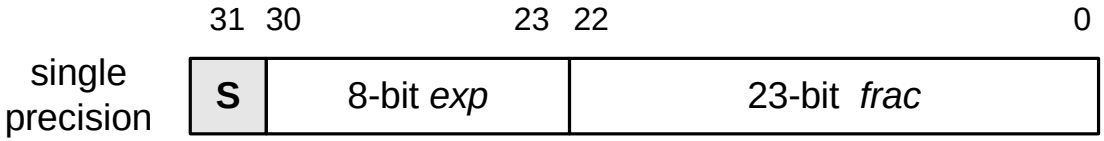
Move binary point 3 places to left: $1.0101_{(2)} \times 2^3$

$10.5 = 1.3125 \times 8$

- Thus by choosing the correct exponent any number can be represented as a fixed point binary number in range 1.0000 - 1.1111 multiplied by an exponent which is a positive or negative power of 2.
- Equivalently, the binary point is "floating"

IEEE-754 32 bit standard floating point

- 32-bit single precision floating point:



$$x = -1^{s} \times 2^{exp-127} \times 1.frac$$

$$5.9 \times 10^{-39} < |x| < 3.4 \times 10^{38}$$

see special cases for smaller numbers

Why not exponent = 128?

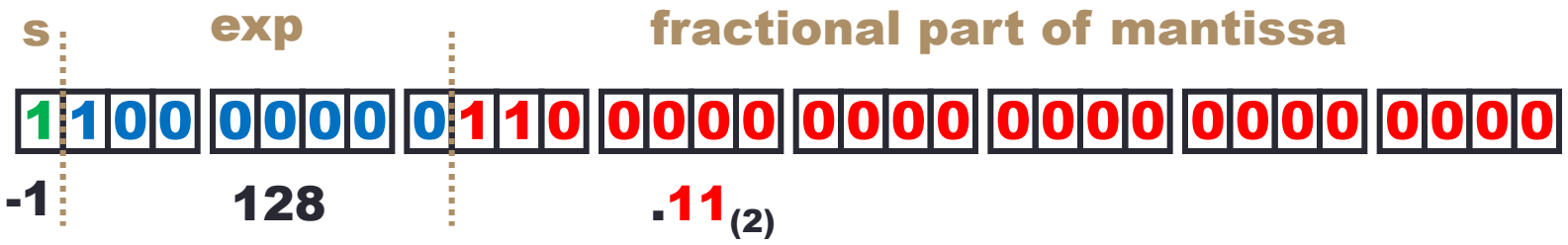
- **Exp field is unsigned**
- MSB *s* is sign-bit: 1 => negative
- Exponent (power of 2) = *exp* - 127

  Note this gives exponent in range [-127,127], and **special case** *exp* = 255
- The MSB of the mantissa is ALWAYS '1', therefore it is not stored
  - mantissa = 1 + *frac**2^{-23} (mantissa = 1.*frac*):

Special cases in IEEE-754 representation

exp	frac	meaning	notes
0	0	+/- 0	The smallest non-zero numbers are used to mean 0, otherwise there is no exact 0!
0	$\neq 0$	Denorm-alised	The mantissa is equal to $2^{-22} frac$ => mantissa in range 2^{-22} - 2
255	0	+/- ∞	The largest exponent is used for special cases
255	$\neq 0$	NaN	not a number, used for 0/0 etc

IEEE-754 → Decimal



- The number above, $C0600000_{(16)}$, must have negative sign,
- *(subtract 127)* Exponent = *exp* -127 = 1,
- *(add 1.)* mantissa = 1+ $0.11_{(2)} = 1.11_{(2)}$

$$-2^1 \times 1.11_{(2)} = -11.1_{(2)} = -3.5$$

Note 1.0 is always added to 0.frac to make a mantissa:

$1.0 \le \text{mantissa} < 2$

Decimal → IEEE 754

- $17.5_{(10)} = 35_{(10)} * 2^{-1} = 10001.1_{(2)} = 2^4 * 1.00011_{(2)}$ ➔

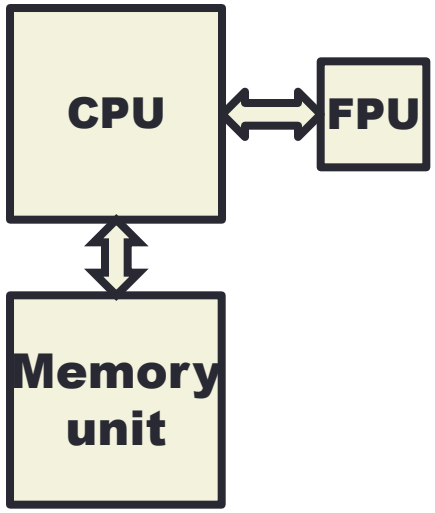
  (add 127) exp = 4+127 = 131 = $10000011_{(2)}$

  (extract bits to right of binary point) frac = $00011000000000000000_{(2)}$

  s = 0 (positive)

  0100 0001 1000 1100 0000 0000 0000 0000

  4 1 8 C 0 0 0 0

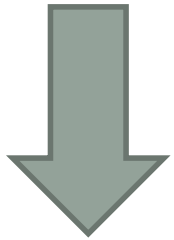


- Floating point arithmetic is typically handled by Floating Point coprocessor (FPU) separate from but connected to the CPU. ARM architecture has FPUs, see latest ARM datasheets for more details.
- We will not consider ARM FPU instructions in this course.

Symbolic Walk-through (SW) in CT1 and CT2

- CT1
  - SW always creates a Par statement equivalent to the code
  - Use SW on any implementation to create an equivalent Par statement
  - Par statements must exactly match for implementation to be correct
    - order of assignments inside Par does not matter
- CT2
  - Specification is simple (a Par)
  - Use SW on implementation to check it works

```
R10 := -(R4+1)
R1 := -(R1+1)
R2 := 0
R3 := hexadecimal 3A
Par {
   R0 := R1 ; R1 := R2 ;
   R2 := R0 ; R11 := R11
}
```



```
Par {
    R10 := -(R4+1)
    R1 := 0
    R0 :=  -(R1+1)
    R2 := R0
    R3 := hexadecimal 3A
}
```

Checking code with Symbolic Walk-through

- **Check code meets specification**
  - Mostly you do this in your head
- **Be careful not to make mistakes**
  - Do it with pencil and paper
  - Test with specific input values

		R0	R1	R2
		r0	r1	r2
MOV	R2, R0	r0	r1	r0
MOV	R0, R1	r1	r1	r0
SUB	R1, R2, R1	r1	r0-r1	r0
ADD	R1, R1, R1	r1	2r0-2r1	r0
ADD	R1, R1, R1	r1	4r0-4r1	r0
RSB	R2, R0, #11	r1	4r0-4r1	11-r1

Conditional branches

see quick reference slide 11

Code	Condition for branch taken	
CMP R0,#0 **BEQ LOOP**	R0 = 0	
CMP R5, R4 **BNE LOOP**	R5 ≠ R4	
CMP R10,#0 **BGE LOOP**	R10 ≥ 0	
CMP R1, #-1 **BGT LOOP**	R1 > -1	
CMP R1, #100 **BLE LOOP**	R1 ≤ 100	
CMP R1, #5 **SUB R1, R1, #1** **BLT LOOP1** **BEQ LOOP2**	R1 < 5: LOOP1 R1 = 5: LOOP2	CMP writes status Status is then tested by BEQ etc Status is unchanged until another CMP

Loops and Labels

Blank lines don't matter

```
START MOV R0, #10
      MOV R1, #0
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
```

```
START
      MOV R0, #10
      MOV R1, #0
LOOP
      SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
```

Analysing Assembly Loops

Loop:

```
Par {
   R0 := R0-1
   R1 := R1+2
}
```

R_0 : contents of R0
R_1 : contents of R1

```
      MOV R0, #10
      MOV R1, #0
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
; what is R0 here?
; what is R1 here?
```

Loop constant:

$$2R_0 + R_1$$

Inside loop

Start and end: $2R_0 + R_1 = 20$

After loop: $R_0 = 0 \Rightarrow R_1 = 20$

Analysing Assembly Loops (2)

Loop:

```
Par {
   R0 := R0-1
   R1 := R1+2
}
```

R_0 : contents of R0
R_1 : contents of R1

```
      MOV R0, #0
      MOV R1, #0
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
; what is R0 here?
; what is R1 here?
```

Loop constant:

$$2R_0 + R_1$$

Inside loop

Start and end: $2R_0 + R_1 = 0$

After loop: $R_0 = 0 \Rightarrow R_1 = 0$

What is the bug?

Analysing Assembly Loops (3)

Loop:

```
Par {
   R0 := R0-1
   R1 := R1+2
}
```

R_0 : contents of R0
R_1 : contents of R1

```
      MOV R0, #10
      MOV R1, #0
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BGE LOOP
; what is R0 here?
; what is R1 here?
```

Loop Constant: $2R_0 + R_1$

Start of Loop: $2R_0 + R_1 = 20 \wedge R_0 \geq 0$ BGE => R0 ≥ 0 when looping

End of Loop: $2R_0 + R_1 = 20 \wedge R_0 \geq -1$ R0 := R0-1 => R0 ≥ -1

After Loop:

BGE => R0 < 0 after loop

$$\begin{array}{rcll} R_0 & < & 0 & \wedge \\ 2R_0 + R_1 & = & 20 & \wedge \\ R_0 & \geq & -1 & \end{array} \quad \Longrightarrow \quad \begin{array}{rcll} R_0 & = & -1 & \wedge \\ R_1 & = & 22 & \end{array}$$

Analysing Assembly Loops (2a)

Loop:

```
Par {
   R0 := R0-1
   R1 := R1+2
}
```

R_0 : contents of R0
R_1 : contents of R1

```
      MOV R0, #0
      MOV R1, #0
      CMP R0, #0
      BEQ LOOPEND
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
LOOPEND
; what is R0 here?
; what is R1 here?
```

Loop constant: $2R_0 + R_1$

Inside loop

Start and end: $2R_0 + R_1 = 0$

After loop: $R_0 = 0 \Rightarrow R_1 = 0$

What type of loop is this?

Analysing Assembly Loops (2b)

```
      MOV R0, #0
      MOV R1, #0
      CMP R0, #0
      BEQ LOOPEND
LOOP  SUB R0, R0, #1
      ADD R1, R1, #2
      CMP R0, #0
      BNE LOOP
LOOPEND
; what is R0 here?
; what is R1 here?
```

```
      MOV R0, #0
      MOV R1, #0
LOOP  CMP R0, #0
      BEQ LOOPEND
      SUB R0, R0, #1
      ADD R1, R1, #2
      B LOOP
LOOPEND
; what is R0 here?
; what is R1 here?
```

While loop optimisation

CT3 Challenge (optional)

- Must it be the same code as my CT3 submission?
  - No
- Must I keep to no more than 50 instructions?
  - Yes
- How fast should my code be?
  - My answer is < 100,000 cycles for 1,999
  - Competitive
- Which instructions can I use?
  - MOV,MVN,ADD,SUB,RSB
  - EOR,AND,ORR
  - B,BEQ,BNE,BGE,BGT,BLE,BLT

ARM Quick Reference

- Compact guide to all of the ARM instruction set that is examined
- Will be needed from CT4 onwards.
- Get used to using it since this reference (and no other) is allowed during tests
- Online links (on the *lectures and handouts* page):
  - Full size
  - Normal size 4 slides per page